

An Intermediate Language and Estimator for Automated Design Space Exploration on FPGAs

Syed Waqar Nabi
School of Computing Science,
University of Glasgow,
Glasgow G12 8QQ, UK.
syed.nabi@glasgow.ac.uk

Wim Vanderbauwhede
School of Computing Science,
University of Glasgow,
Glasgow G12 8QQ, UK.
wim.vanderbauwhede@glasgow.ac.uk

ABSTRACT

We present the TyTra-IR, a new intermediate language intended as a compilation target for high-level language compilers and a front-end for HDL code generators. We develop the requirements of this new language based on the *design-space* of FPGAs that it should be able to express and the *estimation-space* in which each configuration from the design-space should be mappable in an automated design flow. We use a simple kernel to illustrate multiple configurations using the semantics of TyTra-IR. The key novelty of this work is the cost model for resource-costs and throughput for different configurations of interest for a particular kernel. Through the realistic example of a Successive Over-Relaxation kernel implemented both in TyTra-IR and HDL, we demonstrate both the expressiveness of the IR and the accuracy of our cost model.

1. INTRODUCTION

The context for the work in this paper is the *TyTra* project [1] which aims to develop a compiler for heterogeneous platforms for high-performance computing (HPC) that includes many/multi-core CPUs, graphics processors (GPUs) and Field Programmable Gate Arrays (FPGAs). The work we present here relates to raising the programming abstraction for targeting FPGAs, reasoning about its multi-dimensional design space, and estimating parameters of interest of multiple configurations from this design-space via a cost model.

We present a new language, the *TyTra Intermediate Representation* (TIR) language, which has an abstraction level and syntax intentionally similar to the LLVM Intermediate Language [2]. We can derive resource-utilization and performance estimates from TIR code via a light-weight back-end compiler, *TyBEC*, which will also generate the HDL code for the FPGA synthesis tools. We will briefly discuss syntax of the IR and its expressiveness through illustrations, and discuss the cost model we have built around this language.

The TyTra project is predicated on the observation that we have entered a period where performance increases can only come from increased numbers of heterogeneous computational cores and their effective exploitation by software. The specific challenge we are addressing in the TyTra project is how to exploit the parallelism of a given computing plat-

form, e.g. a multicore CPU, GPU or a FPGA, in the best possible way, without having to change the original program. Our proposed approach is to use an advanced type system called Multi-Party Session Types [3] to describe the communication between the tasks that make up a computation, to transform the program using provably correct type transformations, and to use machine learning and a cost model to select the variant of the program best suited to the heterogeneous platform. Our proof-of-concept compiler is being developed and targets FPGA devices, because this type of computing platform is the most different from other platforms and hence the most challenging.

Figure 1 is a concise representation of the compiler's expected flow. The work we present in this paper — identified by the dotted box — is limited to the specification and abstraction of the IR, its utility in representing various configurations, and the cost model built around it which we can use to assess the trade-offs associated with these configuration.

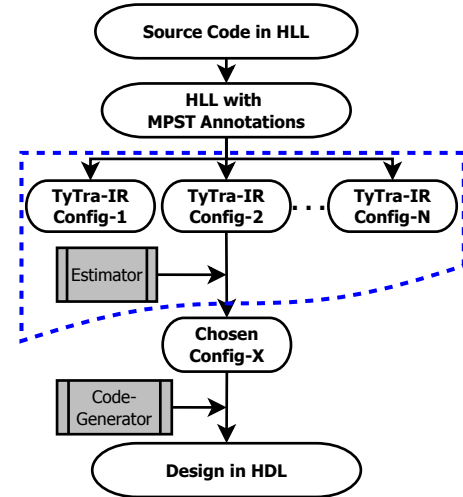


Figure 1: The TyTra project design flow. This paper focuses on the area marked out by dotted lines.

In the next section, we present the TyTra platform model for FPGAs. A very important abstraction for this work is our view of the *design-space* of FPGAs, which we present next, followed by something we call the *estimation-space*. Both the design-space and estimation-space are our attempts to give structure to reasoning around multiple configurations

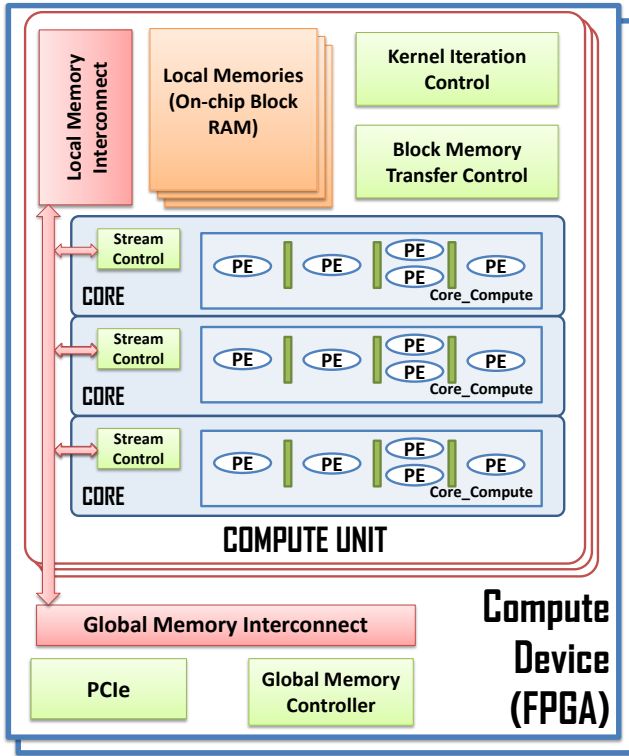


Figure 2: The TyTra-FPGA Platform Model, showing a typical pipelined Core_Compute incorporating ILP, and replicated for thread-parallelism.

of a kernel on an FPGA. We then specify the requirements of the new IR language that we are developing. We follow this with a very brief description of the TIR, and develop this by expressing different FPGA configurations. We then look at the scheme to arrive at various estimates, and evaluate it by a simple example based on the successive-relaxation method. We conclude by briefly discussing some related work and our future lines of investigation.

2. PLATFORM MODEL

The Tytra-FPGA platform model is similar to the platform model introduced by OpenCL [4], but also informed by our prior work on the MORA FPGA programming framework [5], and more nuanced than OpenCL’s to incorporate FPGA-specific architectural features; Altera-OpenCL takes a similar approach [6]. The main departure from the OpenCL model is the *Core* block, and the *Compute-Cores*. Figure 2 is a block diagram of the model, with brief description following. We do however use the terms global memory, local memory, work-group, and work-item, exactly as they are used in the OpenCL framework.

Compute-Device An FPGA device, which would contain one or more *compute-units*.

Compute-Unit Execution unit for a single kernel. An FPGA allows multiple independent kernels to be executed concurrently, though typically there would be a single kernel. The compute-unit contains local memory (block RAM), some custom logic for controlling

iterations of a kernel’s execution and managing block memory transfers, and one or more *cores*.

Core This is the custom design unit created for a kernel. For pipelined implementations, a core may be considered equivalent to a pipeline *lane*. There can be multiple lanes for thread-level parallelism (TLP). The core has control logic for generating data streams from a variety of data sources like local-memory, global-memory, host, or a peer compute-device or compute-unit. These streams are fed to/from the *core-compute* unit inside it, which consists of *processing-elements (PEs)*. A PE can consist of an arithmetic or logic functional unit and its pipeline register, or it can also be a custom scalar or vector instruction processor with its own *private memory*.

3. CONFIGURATION, PERFORMANCE AND COST ABSTRACTIONS

As FPGAs have a fine-grained flexibility, parallelism in the kernel can be exposed by different configurations. It is useful to have some kind of a structure to reason about these configurations; much more so when the goal is an automated design flow. We have created a design-space abstraction for the key differentiating feature of concern of multiple FPGA configurations — the kind and extent of parallelism available in the design. We define an *estimation-space* for capturing the various estimates for a point in the design-space. By defining a design-space, an estimation-space, and a mapping between them, we have a structured approach for mapping a particular kernel to a suitable configuration on the FPGA.

Design Space

The *design-space* is shown in Figure 3. A C2 configuration, on the axis indicating the degree of pipeline parallelism, is a pipelined implementation of the kernel on the FPGA. The other horizontal axis indicates the degree of parallelism achieved by replicating a kernel’s core. This can be done by simultaneously launching multiple calls to a kernel, which is parallelism at a coarse, thread level. Along the same dimension is a medium-grained parallelism, which involves launching multiple work-items of a kernel’s work-group.

A configuration in the xy-plane, C1, has multiple kernel cores, each of which has pipeline parallelism as well. We expect this to be the preferable configuration for most small to medium sized kernels, where the FPGA has enough resources to allow multiple kernel instantiations to reside simultaneously.

Note that we have not explicitly shown the most fine-grained parallelism, i.e., Instruction-Level Parallelism (ILP). The assumption is that it will be exploited wherever possible in the pipeline.

While our current focus is on kernels where we can fit at least one fully laid out custom pipeline on the available FPGA resources, re-use of logic resources is possible for larger kernels by cycling through some instructions in a scalar (C4) or vector (C5) fashion, or by using the run-time reconfiguration capabilities of FPGA devices to load in and out relevant subsets of the kernel implementation (C6).

Finally, C0 represents the generic configuration for any point on the design space.

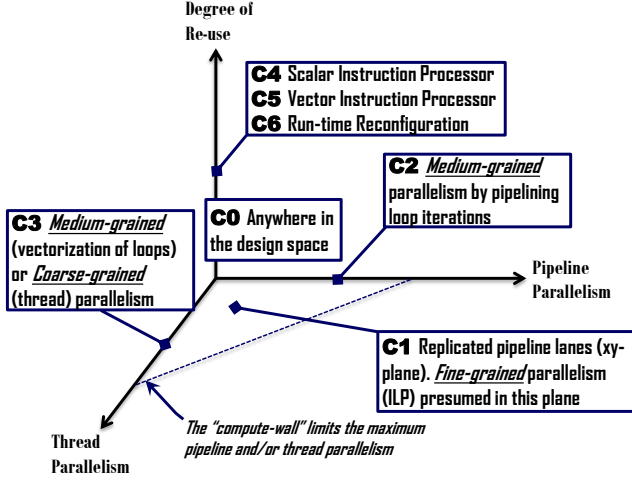


Figure 3: The TyTra-FPGA Design Space Abstraction

Estimation Space

The TyTra design flow (Figure 1) depends on the ability of the compiler to make choices about configurations from the design-space of a particular kernel on an FPGA device. Various parameters will be relevant when making this choice, and the success of the TyTra compiler is predicated on the ability to derive estimates of *reasonable* accuracy for these parameters of concern from the IR, without actually having to generate HDL code and synthesize each configuration on the FPGA. The estimation-space as shown in Figure 4 is a useful abstraction in this context. The obvious aim is to go as high up as possible on the performance axis, while staying within the computation and IO constraint walls.

Having developed the design-space and estimation-space, it follows that the TyTra-IR should intrinsically be capable of working with both these abstractions, as we discuss in the next section.

4. REQUIREMENTS FOR TYTRA-IR

The TyTra-IR is one of the key technologies in our proposed approach, hence its design needs to meet many requirements:

1. Should be intrinsically expressive enough to explore the entire design space of an FPGA (Figure 3), but with a particular focus on custom pipelines because our prime target is HPC applications[7]. (The C1 plane).
2. Should make a convenient target for a front-end compiler that would emit multiple versions of the IR (See Figure 1).
3. Should be able to express access operations in the entire communication hierarchy of the target device¹.
4. Should allow custom number representations to fully utilize the flexibility of FPGAs. If this flexibility offered by FPGAs is not capitalized on, it will be difficult to compete with GPUs for use in HPC for most scientific applications [8].

¹We have omitted the details in this paper, but the TyTra memory-model extends that of LLVM.

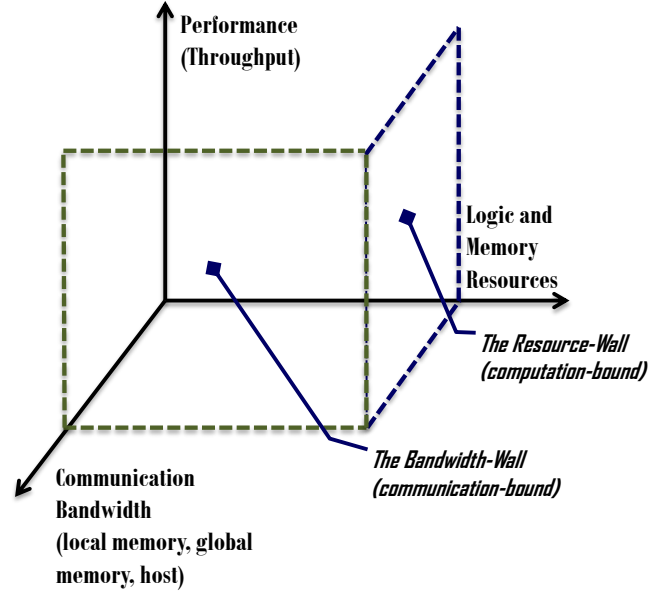


Figure 4: The TyTra-FPGA Estimation Space Abstraction

5. The language should have enough detail of the architecture to allow generation of synthesizable HDL code.
6. A core requirement is to have a light-weight cost-model for high-level estimates. We should be able to place each configuration of interest in the design space (Figure 3) to a point in estimation space (Figure 4).

The above requirements necessitate the design of a custom intermediate language, as none of the existing HLS ("C-to-gates") tools meets all requirements. HLS front-end languages are primarily focused on ease of human-use. High-level hardware description languages like OpenCL or MaxJ[9], having coarse-grained high-level datapath and control instructions and syntactic sugar, are inappropriate as compiler targets because the abstraction level is too high. Moreover, even parallelism friendly high-level languages tend to be constrained to specific types of parallelism, and exploring the entire FPGA design-space would either be impossible, or protracted. The requirements of a lightweight cost-model also motivated us to work on a new language.

5. THE TYTRA-IR

The TyTra-IR (TIR) is a strongly and statically typed language, and all computations are expressed using Static Single Assignments (SSA). The TIR is largely based on the LLVM-IR because it gives us a suitable point of departure for designing our language, where we can re-use the syntax of the LLVM-IR with little or no modification, and allows to explore LLVM optimizations to improve the code generation capabilities of our tool, as e.g. the LegUp [10] tool does. We use LLVM metadata syntax and some custom syntax as an abstraction for FPGA-specific architectural features.

The TIR code for a design has two components:

Manage-IR deals with setting up the streaming data ports for the kernel. It corresponds to the logic in the *core*

outside the *core-compute* (See Figure 2). All Manage-IR statements are wrapped inside the `launch()` method.

Compute-IR describes the datapath logic that maps to the core-compute unit inside the core. It mostly works with very limited data abstractions, namely, streaming and scalar ports. All Compute-IR statements are in the scope of the `main()` function or other functions “called” from it.

By dividing the IR this way, we separate the pure dataflow architecture — working with streaming variables and arithmetic datapath units — from the control and peripheral logic that creates these streams and related memory objects, instantiates required peripherals for the kernel application, and manages the host, peer-device, and peer-unit interfaces. The division between compute-IR and manage-IR directly relates to the division between core-compute unit and the remaining core logic (wrapper) around it (See Figure 2). A detailed discussion of the TIR syntax is outside the scope of this paper, but the following illustration of its use in various configurations gives a good picture.

6. ILLUSTRATION OF IR USE

We use a trivial example and build various configurations for it, to demonstrate the architectural expressiveness of the TIR. The following Fortran loop describes the kernel:

```
do n = 1,ntot
  y(n) = K + ( (a(n)+b(n)) * (c(n)+c(n)) )
end do
```

6.1 Sequential Processing

The baseline configuration, whose redacted TIR code is showed in Figure 5, is simply a sequential processing of all the operations in the loop. This corresponds to C4 configuration in Figure 3.

```
1 ;***** Manage-IR *****
2 define void launch() {
3   @mem_a = addrSpace(3) <NTOT x ui18>, ...
4   @strobj_a = addrSpace(10),
5   !"source", !"mem_a", ...
6   @...[other memory and stream objects]
7   call @main()
8 ;***** Compute-IR *****
9 @main.a = addrSpace(12) ui18,
10 !"istream", !"CONT", !0, !"strobj_a"
11 @...[other ports]
12 define void @f1 (...args...) seq {
13   ui18 %1 = add ui18 %a, %b
14   ui18 %2 = add ui18 %c, %c
15   ui18 %3 = mul ui18 %1, %2
16   ui18 %y = add ui18 %3, @k
17 define void @main () {
18   call @f1(...args...) seq }
```

Figure 5: TyTra-IR code for a sequential processing configuration of a simple kernel

The manage-IR consists of the launch method which sets up the *memory-objects*, which are abstractions for any object that can be the source or destination of streaming data. In this case, the memory object (Figure 5, line 3) is a local-memory instance, indicated by the argument to `addrSpace` qualifier. The stream-objects connect to memory-objects to create streams of data, as shown in lines 4–5. The creation

of streams from memory is equivalent reading from an array in a loop, hence we see that the loop over work-items in Fortran disappears in the TIR. After setting up all stream and memory objects, the main function is called.

The compute-IR sets up the ports (lines 9–11), which are mapped to a stream-object, creating data streams for the compute-IR functions. The SSA datapath instructions in function `f1` are configured for sequential execution on the FPGA, indicated by the keyword `seq`, and then `f1` is called by `main`. Figure 6 shows the block diagram for this configuration.

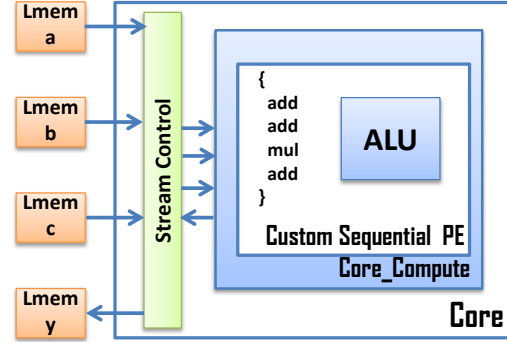


Figure 6: Sequential configuration

6.2 Single Kernel Execution Pipeline

This configuration (C2) is a fully pipelined version of the kernel, and the TIR code is shown in Figure 7.

```
1 @main.a = addrSpace(12) ui18,
2 !"istream", !"CONT", !0, !"strobj_a"
3 @...[other ports]
4 define void @f1(...args...) par {
5   ui18 %1 = add ui18 %a, %b
6   ui18 %2 = add ui18 %c, %c
7 define void @f2 (...args...) pipe {
8   call @f1 (...args...) par
9   ui18 %3 = mul ui18 %1, %2
10  ui18 %y = add ui18 %3, @k
11 define void @main () {
12   call @f2(...args...) pipe }
```

Figure 7: TyTra-IR code for a pipelined configuration of a simple kernel

Note that the available ILP (the two add operations can be done in parallel) is exploited by explicitly wrapping the two instructions into a `par` function `f1`, and then calling it in the pipeline function `f2`. Our prototype parser can also automatically check for dependencies in a pipe function and schedule instructions using a simple as-soon-as-possible policy. See Figure 8 for the block diagram of this configuration.

6.3 Multiple Kernel Execution Pipelines

For simple kernels where enough space is left after creating one pipeline core for its execution, we can instantiate multiple identical pipeline *lanes* (C1). The code in Figure 9 illustrates how this can be specified in TIR. We do not reproduce segments that have appeared in previous listings. See Figure 10 for the block diagram of this configuration.

Comparing with the previous single-pipeline configuration, note that we have a new `par` function `f3` calling the

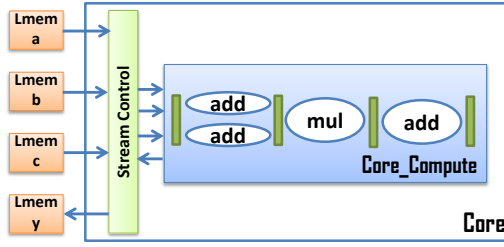


Figure 8: Single Pipeline with ILP

```

1 @main.a_01 = ...
2 @main.a_02 = ...
3 @...[other ports]
4 define void @f1(...args...) par ...
5 define void @f2 (...args...) pipe ...
6 define void @f3 (...args...) par {
7   call @f2(...args...) pipe
8   call @f2(...args...) pipe
9   call @f2(...args...) pipe
10  call @f2(...args...) pipe }
11 define void @main () {
12  call @f3(...args...) par }

```

Figure 9: TyTra-IR code for replicated pipeline configuration of a simple kernel

same `pipe` function four times, indicating replication. Similarly, there are now four separate ports for each array input, and there are four separate streaming objects (not shown) for each of these ports, all of which connect to the same memory object, indicating a multi-port memory.

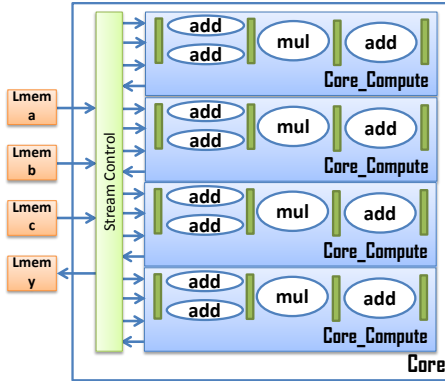


Figure 10: Replicated Pipelines

6.4 Multiple Sequential Processing Elements - Vector Processing

There is one more interesting configuration we can express in TIR by wrapping multiple calls to a `seq` function in a `par` function. This would represent a vectorized sequential processor (C5).

The TIR for this configuration is shown in Figure 11, with only the relevant new bits emphasized.

See Figure 12 for the block diagram of this configuration.

Comparing with the single sequential processor configuration, note that we have a new `par` function `f2` that calls the same `seq` function four times, indicating a replication of

```

1 @main.a_01 = ...
2 ...
3 define void @f1(...args...) seq ...
4 define void @f2 (...args...) par {
5   call @f1(...args...) seq
6   call @f1(...args...) seq
7   call @f1(...args...) seq
8   call @f1(...args...) seq }
9 define void @main () {
10  call @f2(...args...) par }

```

Figure 11: TyTra-IR code for vectorized sequential processing of a simple kernel

the sequential processor.

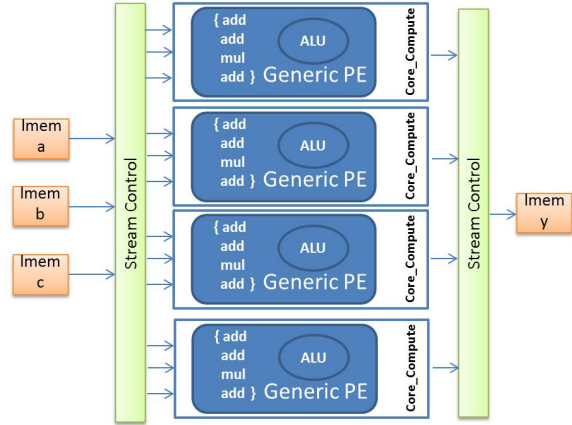


Figure 12: Configuration 4: Vectorized Sequential Processing

7. THE TYTRA-FPGA COST MODEL

Following from the requirement of the ability to get cost and performance estimates as discussed in §4, we designed the TIR specifically to allow generation of accurate estimates. Our prototype TyTra Back-end Compiler (TyBEC) can calculate estimates directly from the TIR without any further synthesis. Many different configurations for the same kernel can be compared by a programmer or – eventually – by a front-end compiler.

Two key estimates are calculated by the TyBEC estimator: the resource utilization for a specific Altera FPGA device (ALUTs, REGs, Block-RAM, DSPs), and the throughput estimate for the kernel under consideration. With reference to Figure 4, this covers two dimensions. An estimate of IO bandwidth requirements is on-going work. For the purpose of this work we make the simplifying assumption that all kernels are compute-bound rather than IO-bound.

7.1 Estimating Throughput

We have described a performance measure called the **EWGT (Effective Work-Group Throughput)** for comparing how fast a kernel executes across different design points. This may be defined as the number of times an entire work-group (the loop over all the work-items in the index-space) of a kernel is executed every second. Measuring throughput at this granularity rather than the more conventional bits-per-second unit allows us to reason about performance at

a coarse enough level to take into account parameters like dynamic reconfiguration penalty. Following is the generic expression which applies to the entire design space (i.e. the C0 root configuration), and specialized expressions for configurations of interest can be derived from it:

$$EWGT = \frac{L \cdot D_V}{N_R \cdot \{T_R + N_I \cdot N_{to} \cdot T \cdot (P + I)\}}$$

Where:

$EWGT$ = Effective Workgroup Throughput

L = Number of identical *lanes*

D_V = Degree of vectorization

N_R = Number of FPGA configurations needed to execute the entire kernel

T_R = Time taken to reconfigure FPGA.

N_I = Number of equivalent FLOP instructions delegated to the average instruction processor

N_{TO} = Ticks taken by one FLOP operation, i.e. CPI.

T = FPGA clock period.

P = Pipeline depth.

I = Number of work-items in the kernel loop.

The key novelty is that the TIR through its constrained syntax at a particular abstraction *exposes* the parameters that make up the expression, and a simple parser can extract them from the TIR code, as we will show in §7.3. If we were to use a higher-abstraction HLS language as our internal IR representation, we would not be able to use the above expression, and some kind of heuristic would have to be involved in making the estimates.

All specialized expressions for different types of configurations can be obtained from the generic expression as follows:

For **C1**, with multiple kernel pipelines, no sequential processing, we set $N_R = 1, T_R = 0, N_I = 1, D_V = 1$, giving us:

$$EWGT = \frac{L}{N_{to} \cdot T \cdot (P + I)}$$

For **C2**, limited to one pipeline lane, setting $N_R = 1, T_R = 0, N_I = 1, D_V = 1, L = 1$ leads to:

$$EWGT = \frac{1}{N_{to} \cdot T \cdot (P + I)}$$

For **C3**, with no pipeline parallelism, we set $N_R = 1, T_R = 0, N_I = 1, D_V = 1, P = 1$ to give:

$$EWGT = \frac{L}{N_{to} \cdot T \cdot I}$$

For **C4**, where PEs are scalar instruction processors, setting $N_R = 1, T_R = 0, D_V = 1$ leads to:

$$EWGT = \frac{L}{N_I \cdot N_{to} \cdot T \cdot (P + I)}$$

For **C5**, where PEs are vector instruction processors, we set $N_R = 1, T_R = 0$, getting:

$$EWGT = \frac{L \cdot D_V}{N_I \cdot N_{to} \cdot T \cdot (P + I)}$$

Finally, for **C6**, with multiple run-time configurations the expression remains the same as **C0**.

As an example, the single-pipelined version in §6.2 corresponds to C2, and multi-pipeline in §6.3, corresponds to C1. We estimated their EWGT based on the relevant expression above, and then compared it to the figures from HDL

simulation. See the comparison in the last row of Table 1. Note that the **cycles/kernel** estimate (second-last row) is very accurate; the somewhat higher deviation of about 20% in EWGT estimate is due to the deviation in estimation of device frequency.

7.2 Estimating Utilization of FPGA Resources

Each instructions can be assigned a resource cost by one of two methods:

1. Use a simple analytical expression developed specifically for the device based on experiments. We have found that the regularity of FPGA fabric allows some very simple first or second order expressions to be built up for most instructions based on a few experiments. The details are outside the scope of this paper.
2. Lookup, and possibly interpolate, from a cost database for the specific token and data type.

The resource costs are then accumulated based on the structural information available in the TIR. For example, two instructions in a **pipe** function will incur additional cost of pipeline registers, and instruction in a **seq** block will save some resources by re-use of functional units, but there will be an additional cost of storing the instructions, and creating control logic to sequence them on the shared functional units. Both the cost and performance estimates follow trivially once we have the kernel expressed in the TIR abstraction.

7.3 The TyTra Estimator Flow

We have written a TyTra Back-end Compiler (TyBEC) that generates estimates as described in this section. Figure 13 shows the flow of the TyBEC.

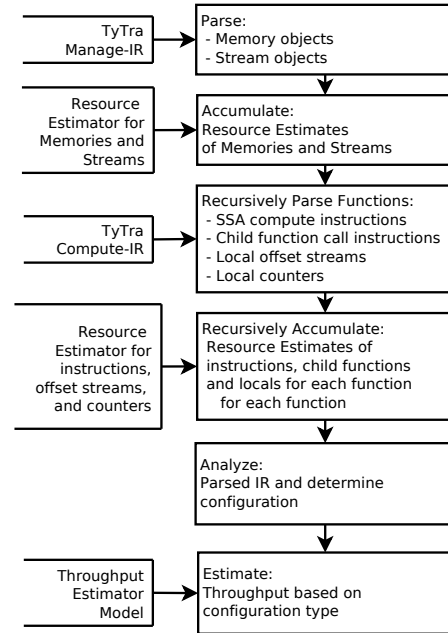


Figure 13: TyTra Back-end Compiler's Flow for Generating Estimates

Using the illustration from §6 we compared the estimates generated by TyBEC, with the actual resource consumption

figures from synthesis of hand-crafted HDL. We only compare the two more relevant example for an FPGA, that is, a single pipeline configuration, and one where pipeline is replicated four times (C2 and C1). The results of comparison are in Table 1. Note that the purpose of these estimates primarily is to choose between different configurations of a kernel. The estimates are quite accurate and well within the tolerance set by the requirements.

Parameter	C2(E)	C2(A)	C1(E)	C1(A)
ALUTs	82	83	36.3K	37.6K
REGs	172	177	18.6K	19.1K
BRAM(bits)	7.20K	7.27K	216K	221K
DSPs	1	1	4	4
Cycles/Kernel	1003	1008	250	258
EWGT	249K	292K	997K	826K

Table 1: Estimated (E) vs actual (A) cost and throughput for C2 and C1 configurations of a very simple kernel

8. CASE STUDY - SUCCESSIVE RELAXATION

We discuss a more realistic kernel in this section, to demonstrate the expressibility of the TIR and effectiveness of its cost model. The *successive over-relaxation method* is a way of solving a linear system of equations, and requires taking a weighted average of neighbouring elements over successive iterations². The listing in Figure 14 is a C-style pseudo-code of the algorithm:

```

1 void onestep(*out,*in) {
2   for(i=1; i<SIZE-1; i++)
3     for(j=1; j<SIZE-1; j++)
4       out[i,j]=( in[i+1,j] + in[i-1,j]
5                 + in[i,j+1] + in[i,j-1]) / 4; }
6 void relax(*a, steps) {
7   for (k=0; k<steps; k++) {
8     onestep(*a_next,*a);
9     a = a_next; } }

```

Figure 14: C code for the successive relaxation algorithm

Figure 15 shows how this translates to TyTra-IR configured as a single pipeline (C2). Note the use of stream offsets (line 21), repeated call to kernel through the **repeat** keyword (line 4), and use of a function of type **comb** (line 12), which translates to a single-cycle combinatorial block. We also use nested counters for indexing the 2D index-space (lines 23-24).

We also implemented this kernel as another configuration with replicated pipelines (C1, similar to the configuration in §6.3).

Results of Estimator

We ran the TyBEC estimator on the two configurations and compared the resource and throughput figures obtained from hand-crafted HDL. Table 2 shows this comparison.

²The TIR has the semantics for standard and custom floating-point representation but the compiler does not yet support floats.

```

1 ; ***** MANAGE_IR *****
2 @launch() {
3   ; define memory and streams objects
4   repeat k=1:NKITER {
5     call @main()
6     @mem_a = @mem_y, !"tir.lmem.copy", ... }
7 ;***** COMPUTE_IR *****
8 ; define ports ...
9 @f1(...args...) par {
10  ui32 %1 = add ui32 %a_e, %a_w
11  ui32 %2 = add ui32 %a_n, %a_s }
12 @f2(...args...) comb {
13  ;[logic instructions for checking boundary condition]
14  ui18 %y = select i1 %11, ui18 %a, ui18 %4
15  @f3(...args...) pipe {
16    call @f1(...args...) par
17    ui18 %3 = add ui18 %1, %2
18    ui18 %4 = udiv ui18 %3, 4
19    call @f2(...args...) comb }
20  @main () {
21    %a_e = ui18 @main.a, !tir.stream.offset, !+1
22    %a_w = ...
23    %ix = ui10 0, !"counter", !(NCOLS-1)
24    %iy = ui10 0, !"counter", !(NROWS-1), !"ix"
25    call @f3(...args...) pipe }

```

Figure 15: TyTra-IR code for the relaxation kernel configured as a single pipeline

Resource	C2(E)	C2(A)	C1(E)	C1(A)
ALUTs	528	546	5764	5837
REGs	534	575	4504	4892
BRAM(bits)	5418	5400	11304	11250
DSPs	0	0	0	0
Cycles/Kernel	292	308	180	185
EWGT	57K	43K	92K	72K

Table 2: Estimated (E) vs actual (A) cost and throughput for two configurations C2 and C1 of relaxation kernel

A reasonable accuracy of the estimator is clearly indicated by these comparisons. This vindicates our observation that an IR designed at an appropriate abstraction will yield estimates of cost and performance in a very straightforward and light-weight manner, that are accurate enough to make design decisions. Hence it is our plan to use this IR to develop a compiler that takes legacy code, and automatically compares various possible configurations on the FPGA to arrive at the best solution.

9. RELATED WORK

High-Level Synthesis for FPGAs is an established technology both in the academia and research. There are two ways of comparing our work with others. If we look at the entire TyTra flow as shown in Figure 1, then the comparison would be against other C-to-gates tools that can work with legacy code and generate FPGA implementation code from it. As an example, LegUP[10] is an upcoming tool developed in the academia for this purpose. Our own front-end compiler is a work in progress and is not the focus of this paper.

A more appropriate comparison for this paper would be to take the TyTra-IR as a custom language that allows one to program FPGAs at a higher abstraction than HDL, and could be used as an automated or manual route to FPGA programming. Reference [9] for example discussed the Chimps language that is at a similar abstraction as TIR and generates HDL description. Our work is relatively less developed compared to Chimps, however there is nothing equivalent to the estimator model that we have in the Ty-

BEC. The MaxJ is a Java-based custom language used to program Maxeler DFEs (FPGAs) [11]. It is in some ways similar to TIR in the way uses stream abstractions for data, creates pipelines by default. In fact, our IR has been informed a study of the MaxJ language. The use of streaming and scalar ports, offset streams, nested counters, and separation of management and computation code in the TIR is very similar to MaxJ. However, the similarity does not extend much beyond these elements. TIR and MaxJ are at very different abstraction levels, with the latter positioned to provide a programmer-friendly way to program FPGAs. The TIR on the other hand is meant to be a target language for a front-end compiler, and is therefore lower abstraction and fine-grained. This fine-grained nature allows a much better observability and controllability of the configuration on the FPGA, which makes it a more suitable language to explore the entire FPGA design space.

Altera-OCL is an OpenCL compatible development environment for targeting Altera FPGAs[6]. It offers a familiar development eco-system to programmers already used to programming GPUs and many/multi-cores using OpenCL. A comparison of a high-level language like OpenCL with TyTra-IR would come to similar conclusions as arrived in relation to MaxJ. In addition, we feel that the intrinsic parallelism model of OpenCL, which is based on multi-threaded work-items, is not suitable for FPGA targets which offer the best performance via the use of deep, custom pipelines. Altera-OCL is however of considerable importance to our work, as we do not plan to develop our own host-API, or the board-package for dealing with FPGA peripheral functionality. We will wrap our custom HDL inside an OpenCL device abstraction, and will use OpenCL API calls for launching kernels and all host-device interactions.

10. CONCLUSION AND FUTURE WORK

In this paper, we showed that the TIR syntax makes it very easy to construct a variety of configurations on the FPGA. While the current semantics are already reasonably expressive, the TIR will evolve to encompass more complex scientific kernels. We showed that we can estimate very accurate estimates of cost and performance from the TIR without any further translations. We indicated that automatic HDL generation is a straightforward process, which is a work in progress, and our immediate next step.

We plan to improve the accuracy of the estimator with better mathematical models. We will also make our estimator tool more generic, as for this proof-of-concept work the supported set of instructions and data-types is quite limited. The compiler will also be extended to incorporate optimizations, in particular we aim to incorporate LegUP's sophisticated LLVM optimizations before emitting HDL code[10].

While we work on maturing the TIR and the back-end compiler, we will be moving higher up the abstraction as well. We will be investigating the automatic generation of the TIR from HLL, with the help of Multi-Party Session Types to ensure correctness of transformations.

Acknowledgement

The authors acknowledge the support of the EPSRC for the TyTra project (EP/L00058X/1).

11. REFERENCES

- [1] "The TyTra project website," <http://tytra.org.uk/>, accessed: 2015-04-17.
- [2] Chris Lattner and Vikram Adve, "The LLVM Instruction Set and Compilation Strategy," CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS-R-2002-2292, Aug 2002.
- [3] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," *SIGPLAN Not.*, vol. 43, no. 1, pp. 273–284, Jan. 2008. [Online]. Available: <http://doi.acm.org/10.1145/1328897.1328472>
- [4] J. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010.
- [5] S. R. Chalamalasetti, S. Purohit, M. Margala, and W. Vanderbauwhede, "Mora-an architecture and programming model for a resource efficient coarse grained reconfigurable processor," in *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*. IEEE, 2009, pp. 389–396.
- [6] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh, "From opencl to high-performance hardware on fpgas," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 531–534.
- [7] W. Vanderbauwhede, "On the capability and achievable performance of fpgas for hpc applications," in *Emerging Technologies Conference*, Apr 2014.
- [8] M. Vestias and H. Neto, "Trends of cpu, gpu and fpga for high-performance computing," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014, pp. 1–6.
- [9] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers, "Chimps: A c-level compilation flow for hybrid cpu-fpga architectures," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, Sept 2008, pp. 173–178.
- [10] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. [Online]. Available: <http://doi.acm.org/10.1145/1950413.1950423>
- [11] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *Computing in Science Engineering*, vol. 14, no. 4, pp. 98–103, July 2012.